A3P 2013/2014 - Groupe 4D
Rapport de conception du jeu
« Amélia »

Page du jeu : http://www.esiee.fr/~pallott/a3p/

I) Informations sur le jeu

A) **Auteurs** :

DA ROCHA Olivier
PALLOT Timothée

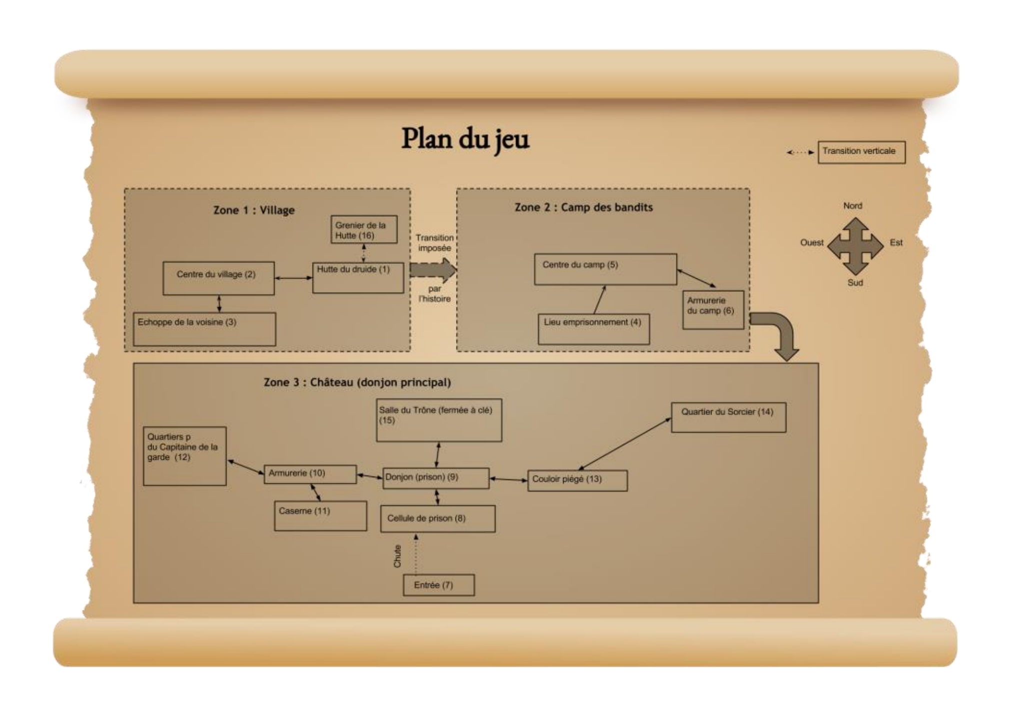
B) **Thème** :

Dans un univers médiéval-fantastique, la fille d'un druide doit sauver sa peau après son enlèvement.

C) **Résumé du scénario**

Amélia, fille du druide Oramic, vit au village de Sous-les-Feuilles. Elle y est un jour capturée par des bandits dont elle parvient à se défaire. Le commanditaire de l'enlèvement s'avère être le régent. Amélia se rend alors au château pour le défaire ; elle y apprendra qu'elle est l'héritière de la famille royale disparue. Le régent vaincu, elle monte sur le trône.

D) **Plan** (*à consulter en haute définition sur la page du projet*)



E) **Scénario détaillé**

Zone 1 : Village de Sous-les-Feuilles

Le village comporte **quatre** salles : la hutte du druide, la chambre d'Amélia, le centre du village et le magasin. On peut y rencontrer des personnages et y trouver des objets, sans incidence directe s'ils sont laissés. Au magasin se trouve un chat qui suivra le joueur pendant toute l'aventure. Amélia est enlevée si elle sort du village par le **nord**.

Zone 2 : Camp des bandits

Il comporte **trois** salles. La première est une prison dont on peut sortir en ramassant un **sac** à proximité. La seconde est le camp à proprement parler, dont la sortie ouest donne sur le château. La troisième est une armurerie. On trouve des objets dans le camp, sans incidence directe s'ils ne sont pas ramassés.

Zone 3 : Château

Il contient **neuf** salles. L'entrée donne, au nord, sur un précipice : le joueur se retrouve dans une cellule. Il peut ensuite se déplacer librement dans six autres salles pour y trouver les **deux clés** qui ouvrent la salle du trône, au nord. Dans celle-ci, il faut **attaquer** le régent puis lui **prendre le sceptre** pour gagner la partie.

F) **Détail des items et personnages**

Les items ayant une influence sur le déroulement du jeu sont :

- « parchemin » : répondant à l'exercice 7.44 (**Beamer**), ce parchemin permet de se téléporter (avec la commande *utiliser parchemin*) dans la salle où on l'aura chargé (commande *charger*). On le trouve dans les quartiers du sorcier.
- « gâteau » : répondant à l'exercice 7.34 (**magic cookie**), ce gâteau augmente de 150 onces le poids maximal porté par le joueur lorsqu'il est consommé (commande *manger*). On le trouve dans les quartiers du capitaine.
- « sac » : ce sac trouvé dans le camp des bandits permet, grâce aux objets qu'il contient, de sortir de la prison où se trouve le joueur.
- Les deux clés trouvées respectivement dans les quartiers du sorcier et ceux du capitaine permettent d'ouvrir la salle du trône, pour accéder à la scène finale du jeu. Elles répondent ainsi à l'exercice facultatif 7.45 (**locked door**).
- « sceptre » : il s'agit du sceptre royal qui, lorsqu'il est saisi, entraîne la fin du jeu. Il se trouve naturellement dans la salle du trône.

Les items n'ayant pas d'influence directe sur le jeu sont :

- « provisions » : trouvées dans la hutte du druide.
- « bâton » : trouvé dans la hutte du druide.
- « médaillon » : trouvé dans la chambre d'Amélia.
- « tunique » : trouvée dans la chambre d'Amélia.
- « affaires » : trouvées dans l'armurerie du camp.
- « armes » : trouvées dans l'armurerie du château.

Les différents personnages sont :

- Oramic, le druide. On le trouve dans sa hutte, où il adresse la parole à Amélia.
- Tassia, la vendeuse. On la trouve dans son magasin, où elle adresse la parole à Amélia.
- Le chat. On le trouve dans le magasin. Il suit ensuite le joueur tout au long de l'aventure, et répond donc à l'exercice 7.49 (**moving Character**).
- Le capitaine de la garde. On le trouve dans ses quartiers, où on peut l'entendre parler.
- Les gardes. On les trouve dans la caserne, où on peut les entendre parler.
- Le sorcier. On le trouve dans ses quartiers, où on peut l'entendre parler.
- Le régent. On le trouve dans la salle du trône, où il s'adresse à Amélia.

G) **Situations gagnantes et perdantes**

Il existe une unique façon de perdre : si le temps est écoulé. Il s'agit en fait d'un nombre maximum de déplacements, fixé à 35 lors de l'exercice 7.42 (**time limit**). Il existe, de même, une unique façon de gagner : en venant à bout du régent puis en s'emparant du sceptre royal.

II) Réponses aux exercices

7.10

La fonction *getExitString()* retourne une chaîne de caractères de la forme :
"Sorties : nord ouest" où nord et ouest sont les deux seules sorties possibles de la salle courante.
La chaîne retournée est la variable locale vExits.
On utilise la méthode *keySet()* de la HashMap aExits (qui contient les sorties possibles de chaque salle). Celle-ci retourne un ensemble (Set) vKeys contenant les noms des sorties de cette salle sous forme de chaîne de caractères.
On parcourt cet ensemble à l'aide d'une boucle *for each* (for(String vParcours : vKeys)). La variable vParcours va prendre toutes les valeurs de l'ensemble vKeys. On les ajoute à vExits qui sera la String retournée.

7.14

On devra tout de même changer la classe Game si on ajoute une nouvelle commande. Il faudra créer une nouvelle méthode pour indiquer le comportement de cette commande, et ajouter à *processCommand()* un cas pour que cette commande puisse être comprise et appelée.

7.18.8

Nous avons incorporé des boutons en utilisant des objets JButton. Les boutons ont été disposés en GridLayout (en ajoutant un GridLayout au sein du BorderLayout préexistant) et leurs étiquettes ont été ajoutées en passant une String en paramètre au constructeur de chaque bouton. (ex : *this.aButton = new Button("Texte");*).
Les boutons, contenus dans un tableau aButton[], sont ajoutés à la fenêtre en utilisant la méthode *add(aComponent)* de l'objet JPanel correspondant à notre grille de boutons (vGrid) en mettant chaque bouton en paramètre : *add(aButton[i])*.
Afin de leur donner une utilité, on ajoute aux boutons un ActionListener (interface implémentée par UserInterface). L'action correspondant à chaque bouton est décrite dans la procédure *actionPerformed()* (imposée par l'interface). Par exemple si on clique sur le bouton indiquant "Nord", l'action qui sera exécutée est la même que si le joueur avait tapé "aller nord".

7.21

Les informations concernant un Item devront être produites dans la classe Item de sorte que chaque objet de cette classe contiennent les informations de l'Item qu'il représente.
La String décrivant l'Item doit être produite par la classe Room car c'est elle qui s'occupe de produire la String décrivant le lieu. La classe GameEngine affichant déjà la description du lieu, c'est également elle qui affichera la description de l'Item.
Notre classe Item contient aussi des fonctions statiques créant des objets Item (leur donnant tous leurs attributs : nom, description, poids) et les retournant, afin de décharger la classe GameEngine.

7.23

La commande retour permettant de revenir à la Room précédente, et ce jusqu'au début du jeu, n'était pas en adéquation avec le scénario de notre jeu, nous avons choisi d'interdire au joueur de revenir en arrière après avoir passé certains points du jeu.

7.26

La pile (Stack) permet de stocker les lieux précédemment visités en incorporant le dernier lieu (grâce à la méthode push()) en haut de la pile. C'est donc lui qui ressortira en premier lorsque qu'on l'appelera avec la méthode pop() lors de l'utilisation de la commande retour.

7.29
La classe Player permet d'alléger le code de la classe GameEngine en y stockant la variable aCurrentRoom (correspondant au lieu dans laquelle se trouve le joueur, donc propre à chaque joueur) et aPreviousRoom (correspondant à la Stack de lieux précédemment visitées, donc propre au parcous de chaque joueur).

7.30
Les commandes prendre et lâcher permettent au joueur de prendre un objet et de le relâcher.
La méthode take() (permettant de prendre un objet) ajoute l'Item désigné à l'inventaire (implémenté à l'exercice 7.33) du joueur et supprime ce même Item de la Room dans laquelle il se trouve.
La méthode drop() (permettant de lâcher un objet) ajoute l'Item désigné à la salle actuelle (pas forcément le lieu initial de l'Item) et le supprime de l'inventaire du joueur.

7.31.1
La classe ItemList permet, à l'aide d'une HashMap, de contenir des objets Item. Cela est très pratique notamment dans les classes Room et Player. Dans la classe Room, cela crée une liste des objets présents dans le lieu. Dans la classe Player, cela sert d'inventaire et contient tous les objets ramassés par le joueur.
La classe ItemList contient plusieurs méthodes très pratiques venant de la HashMap, comme containsItem(), add(), getItemWeight(), etc.

7.32
Pour implémenter un poids maximal dans notre jeu, nous avons décidé de créer un attribut (de type int) que nous avons initialisé avec une valeur arbitraire. Chaque objet Item ayant un int correspondant au poids de l'Item. Lorsque l'on ramasse un objet et que le poids de cet objet ajouté au poids des autres excède la limite choisie, il est impossible de ramasser le dit objet.

7.33
Comme il est déjà dit au *7.31.1* l'inventaire du joueur est géré par la classe ItemList qui contient, dans son attribut HashMap, les Items ramassés par le joueur, et offre des méthodes permettant toutes sortes de manipulations sur l'état de l'inventaire (ajouter des Items, savoir si un Item est dans l'inventaire, etc.).
Lorsque la commande inventaire est tapée, une fenêtre répertoriant les objets Item de l'inventaire est affichée. On y trouve également la description et le poids de l'Item.

7.34
Le gâteau magique (magic cookie) agit lorsque la commande manger est tapée (et que le gâteau est dans l'inventaire). Lorsque cette commande est tapée, le poids maximal est modifié par un poids arbitrairement défini grâce à la méthode setMaxWeight().

7.35 et 7.41.1
L'énumération *CommandWord* décrit les mots de commandes possibles. Chaque élément de l'*enum* a un équivalent « String » qui correspond au mot de commande tapé par le joueur.
La classe CommandWords sert maintenant à créer une HashMap contenant toutes les commandes sous forme d'éléments de l'*enum*, avec leurs équivalents String comme clés.

7.35.1
Grâce à l'implémentation de l'énumération CommandWord, l'utilisation d'un switch dans interpretCommand() est possible et plus claire. Lorsqu'une commande est tapée, interpretCommand() la traite sous forme d'un CommandWord et d'un second mot (String).

7.42
La limite de temps que nous avons implémentée dans notre jeu correspond au nombre déplacements que le joueur peut effectuer avant que la partie ne se finisse. A chaque changement de lieu, l'attribut aTime (avec une valeur arbitrairement choisie) est décrémenté. Si cet attribut se retrouve à 0, le jeu s'arrête.

7.42.2
Nous avons décidé de garder l'interface graphique de base.

7.43
Plusieurs « trap doors » ont été implémentées dans le jeu pour les besoins du scénario. Il y a 5 salles dans lesquelles on se retrouve dans l'incapacité de revenir en arrière (la cellule dans le camp de bandit, le centre du camp, l'entrée du château, la cellule du château et la salle du trône). Lorsque le joueur entre dans l'une de ces salles, la pile contenant les salles précédemment visitées est alors vidée.

7.44
Pour implémenter le « Beamer » (parchemin de téléportation dans le jeu), la classe *Beamer* héritant de *Item* a été créée. Une fois l'objet Beamer dans l'inventaire, on peut l'utiliser en le chargeant, grâce à la commande « *charger* », ce qui mémorise le lieu dans lequel on le charge dans l'attribut aMemRoom.
Pour activer le parchemin il faut taper « *utiliser parchemin* », ce qui transporte le joueur dans la pièce qu'il a précédemment chargé. Le changement de pièce se fait de la même la même façon que pour un changement de lieu standard.

7.46
La classe TransporterRoom héritant de Room a été limitée à une portée de 8 lieux pour les besoins du scénario. Un numéro pseudo-aléatoire est déterminé par la méthode getRandomRoom() qui renvoie l'objet Room correspondant au numéro déterminé. A la sortie de la TransporterRoom, le joueur est transporté dans la pièce déterminée précédemment.

7.46.1
La commande alea n'est utilisée que pendant les tests et se trouve dans la liste des commandes à effectuer dans les fichiers de tests. La commande alea suivie d'une String contenant un nombre correspondant au numéro d'un lieu (de 1 à 16) permet de stocker ce lieu pour que le joueur y soit transporté à la sortie de la TranporterRoom au lieu d'un lieu quelconque.
La commande alea est appelée au début et à la fin des fichiers de test, respectivement pour l'activer et la désactiver. Elle n'est disponible qu'en **mode test**, c'est pourquoi le moteur passe en mode test lors de la lecture des fichiers test. Le mode test est ensuite désactivé pour qu'alea ne soit pas utilisé dans d'autres circonstances.

7.46.2
Au moment de la lecture de cet exercice, l'héritage avait déjà été employé pour la classe Beamer à l'exercice 7.44. Quant aux exercices 7.43 à 7.45, notre conception ne nécessitait pas de changements de cet ordre.

7.47 (non réalisé)
Cet exercice engendre de larges modifications du moteur de jeu. À la suite de plusieurs essais, nous avons rencontré de nombreux problèmes que nous ne sommes pas parvenus à résoudre. Nous nous sommes ainsi concentrés sur les exercices suivants.

7.47.1
À l'occasion de cet exercice, les packages *pkg_core* (pour le « cœur » du jeu, c'est-à-dire les classes GameEngine et Player entre autres), *pkg_commands*, *pkg_items* et *pkg_rooms* furent créés et les différents imports réalisés. Le projet nous apparut tout de suite plus structuré.

7.48
Pour cet exercice, il fut décidé de créer une classe Character, prenant comme attributs un nom de personnage et un dialogue (tous deux sous forme de String). Une HashMap associant des String (noms des personnages) à des Character fut ensuite ajouté comme attribut d'instance dans la classe Room, afin que chaque salle puisse contenir plusieurs personnages, facilement désignables par leur nom. Plusieurs méthodes furent ajoutées à la classe Room pour interagir avec cette HashMap, puis la méthode getLongDescription() fut modifiée pour qu'elle renvoie la liste des personnages et les dialogues en plus de la liste des objets.

7.49
Pour créer des personnages mobiles, nous avons créé une nouvelle classe héritant de Character. Le schéma de déplacement choisi était « suivre le joueur d'une salle à l'autre ». C'est pourquoi nous avons nommé cette classe « FellowCharacter » : elle gère les « camarades ».
Elle possède un attribut supplémentaire qui est la *salle courante* (aCurRoom) du personnage,

afin qu'il puisse gérer lui-même son déplacement à l'aide d'une méthode *move(Room)* : celle-ci supprime le personnage de sa salle courante, remplace cet attribut par une Room de destination reçue en paramètre, et ajoute le personnage à cette salle (ce qui est rendu possible par l'héritage).
La méthode *move()* est appelée par GameEngine à chaque déplacement du joueur (donc dans la méthode *teleport(Room)* commune à tous les déplacements). Il faut pour cela que l'objet Player renvoie son camarade potentiel, c'est pourquoi un attribut FellowCharacter (avec accesseur et modificateur) furent ajoutés à la classe Player.
De même, il fallait indiquer au Player de prendre un camarade s'il en existe un dans sa salle courante. Une méthode renvoyant l'éventuel FellowCharacter d'une salle fut ajoutée dans ce but dans la classe Room (elle parcourt la HashMap de Character en tentant de les « caster » en FellowCharacter et renvoie le premier Character pour lequel la conversion fonctionne). Cette méthode est ensuite utilisée dans teleport(), afin qu'à l'arrivée dans la salle, le camarade soit automatiquement donné au joueur.

7.53 et **7.54**
Pour ces exercices, nous avons travaillé sur notre code à l'aide de GEdit sur Debian et Notepad++ sur Linux. Nous n'avons pas rencontré de problème pour créer et faire fonctionner la méthode *main*, ni pour compiler (à l'aide de *javac*) et exécuter le jeu *via* la console, puisque nous avions déjà pris l'habitude d'utiliser *javadoc* en console.

58 à **58.2**
Le jar fut créé avec BlueJ, mais nous avons aussi étudié la syntaxe du fichier MANIFEST. Nous avons constaté que l'encodage des fichiers test pouvait poser un problème selon que le jeu était exécuté sur GNU/Linux ou Windows. Grâce à la documentation de la classe Scanner, nous avons pu déclarer dans le code que ces fichiers étaient encodés en UTF-8, et ainsi empêcher une mauvaise lecture du fichier sous Windows.
Un second problème s'est posé : la classe MP3 tentait d'accéder aux musiques à l'extérieur de l'archive jar (si le dossier music se trouvait dans le même répertoire, les musiques étaient lues, mais ce n'était pas le cas s'il était dans l'archive). Des recherches nous ont permis de déduire que les objets File et FileInputStream utilisé par la classe MP3 n'était pas appropriés pour une archive jar ; c'est pourquoi, après avoir consulté la documentation et nous être inspirés du fonctionnement de l'affichage d'images dans notre jeu, nous avons fait appel à la méthode *getResourceAsStream* de ClassLoader pour remplacer le FileInputStream.
Nous avons appliqué la même démarche à la procédure test(), qui accédait aux fichiers tests sous forme de File, et empêchait donc leur lecture lorsqu'ils sont contenus dans l'archive jar.
Les musiques et fichiers tests contenues dans le jar sont à présent lus sans problème.

III) Mode d'emploi

Pour s'exécuter, le jeu nécessite **Java 7**. L'archive Amelia.jar peut être exécutée à l'aide de la commande suivante, tapée dans un terminal :

java -jar Amelia.jar

L'objectif du jeu est de progresser jusqu'à la salle du trône. Le joueur est guidé par les indications s'affichant sur la fenêtre de jeu, et la fenêtre d'aide (à ouvrir avec la commande « *aide* »).

Des informations supplémentaires se trouvent dans le fichier README.TXT, mais avant tout sur la page Web du projet, à la section **Aide** (un manuel des commandes y est disponible).

Les commandes possibles sont :

- aller
- retour
- quitter
- prendre
- lâcher
- aide
- manger
- regarder
- jouer/arrêter
- inventaire
- attaquer
- charger
- utiliser
- alea *et* test (*à des fins de test uniquement*)

IV) Déclaration anti-plagiat

Le projet comporte une classe MP3 adaptée du travail de Kevin Wayne (Princeton University), comme indiqué dans la documentation. Cette même classe importe la librairie JLayer 1.0.1 de Javazoom.

Le jeu comporte également des images et morceaux de musique soumis au droit d'auteur. La déclaration complète les concernant se trouve sur la page Web du projet à la section **Copyright**.